Canonical Execution Semantics for Stochastic Program Generators

Thomas Dionysopoulos, CFA

Abstract

AI systems increasingly generate computation—through large language models, program synthesizers, and agent frameworks—rather than having humans write it directly. When the generator is stochastic, independently generated programs that represent the same intended computation arrive in different surface forms, producing different hashes, different provenance records, and failed replay comparisons. We argue that execution systems for stochastic generators require a *canonical execution boundary*: an architectural invariant that partitions the pipeline into a stochastic upstream (generation, admission) and a deterministic downstream (planning, execution, provenance). Stochasticity does not propagate beyond this boundary.

We report on BLISP, a deterministic execution system that enforces this boundary through four mechanisms: (1) *typed specifications* that constrain stochastic generators to propose structured parameter tuples rather than executable expressions, eliminating surface-form divergence by construction; (2) a *canonicalization pipeline* whose traced normalization algebra collapses 278 surface forms to 235 canonical operations; (3) *8-layer execution hashing* that decomposes provenance into distinct semantic layers for per-layer fault localization; and (4) *description/identity separation* in a content-addressed capability registry, where discovery metadata evolves independently of execution identity.

We evaluate canonicalization collapse under 1,200 stochastic LLM generations, replay determinism across 50 independent runs, and provenance stability across registry evolution. The results support the central thesis: AI-generated computation should not execute stochastic outputs directly; stochastic proposals should first compile into deterministic semantic objects before execution begins.

1 Introduction

Programs are increasingly generated by stochastic systems—large language models, program synthesizers, evolutionary search—rather than written by human programmers. This shift changes the requirements on the execution substrate. A human programmer writes one expression and debugs it interactively. A stochastic generator produces many candidate expressions across independent runs, and the system must determine which are equivalent, which have been seen before, and whether a given result can be reproduced.

These requirements are execution-level, not generation-level. They do not concern how the generator produces expressions (its architecture, training, or decoding strategy), but what the execution system does with them afterward. Existing AI agent systems orchestrate stochastic computation but do not provide deterministic execution identity. The central observation is:

> *Execution systems for stochastic generators require a canonical execution boundary: an architectural invariant beyond which stochasticity does not propagate. After this boundary, all downstream artifacts—execution plans, results, hashes, provenance—are deterministic functions of three inputs: the canonical form, the registry state, and the data.*

The canonical execution boundary is the point at which stochastic proposals become deterministic executable objects—concretely, the point at which typed specifications and their canonical morphism expansions replace the generator's surface-form output.

Traditional compilers normalize for optimization: transforming an expression into a form that executes faster, potentially through many valid intermediate representations. Canonicalization for identity transforms an expression into a *unique* normal form such that any two expressions that normalize to the same canonical form produce the same canonical string, the same hash, and the same provenance record. Under human

authorship, where each expression is written once, this distinction is minor. Under stochastic generation, where the same canonical form may be expressed differently on every run, it determines whether execution results can be compared, deduplicated, and replayed.

The divergence problem. Consider a system that asks a stochastic generator to produce a BLISP expression for "rolling z-score of log returns." Independent runs may produce:

Run	Expression
1	`(-> (stdin) (dlog) (wzs 25 1))`
2	`(-> (stdin) (dlog) (rol_zsc 25 1))`
3	`(rol_zsc (dlog (stdin)) 25 1)`
4	`(wzs (dlog (stdin)) 25 1)`

All four normalize to the same canonical form. Without canonicalization, the system sees four distinct programs with four distinct hashes. Downstream, morphism expansion treats them as different candidates, provenance records diverge, and replay checks fail even though all four normalize to the same executable representation. With BLISP's canonicalization pipeline, all four reduce to the same canonical form: `(rol_zsc (dlog (stdin)) 25 1)`, producing one hash, one provenance record, and deterministic replay.

This is not a hypothetical scenario. Any stochastic generator with non-zero temperature produces surface-form variation as a byproduct of sampling. The variation is harmless for human-authored code (a linter catches it, or a reviewer ignores it). For agent-generated computation feeding into deterministic execution, it accumulates as execution divergence: different hashes, different provenance chains, failed replay comparisons.

Design pressures. Human-authored programming systems evolved under a specific set of pressures: readability, stylistic flexibility, interactive debugging, mutable state. Agent-generated computation introduces different ones:

- **Typed specification.** Stochastic generators propose structured parameter tuples—a computational family, a scoring metric, parameter ranges, a data source—over a constrained semantic space. The execution system expands these into executable configurations. The generator never writes the executable expression directly; it operates upstream of the execution boundary.
- **Canonical form.** Multiple surface forms that produce identical execution plans must reduce to a single executable representation.
- **Deterministic replay.** Identical specifications against identical data must produce bit-identical results and hashes.
- **Content-addressed identity.** Capabilities, morphisms, and execution results are identified by their canonical content, not by name or file path.
- **Provenance decomposition.** When two execution hashes differ, the system must localize the divergence to a specific semantic layer without re-execution.
- **Description/identity separation.** Improving how capabilities are described (aliases, tags, documentation) must not change their execution identity or invalidate prior execution hashes.

None of these pressures is unique to agent-generated computation, but collectively they constitute a design regime where existing programming systems provide partial coverage at best. Notebooks provide none. Workflow systems provide replay (with caveats) but not canonical forms. Content-addressed build systems (Nix, Bazel) provide deterministic replay but hash all metadata—descriptions, maintainer fields—into the build identity, coupling interface evolution to execution identity.

Contributions. We report on BLISP, a deterministic execution system designed for stochastic program generators. We make the following contributions:

1. **Typed specification as constrained stochastic proposal.** Stochastic generators produce declarative parameter tuples from a constrained semantic space; the system expands these into canonicalized, content-addressed executable configurations via Cartesian product. Surface-form divergence is eliminated by construction—the generator never writes the executable expression (§3.1).
2. A **canonicalization pipeline** with a traced normalization algebra that collapses stochastic surface-form divergence to canonical executable forms. The pipeline implements three rewrite classes (threading, aliasing, argument reordering) and is measurably effective: 278 accepted surface forms collapse to 235 canonical operations in the static registry (§3.2).
3. **8-layer execution hashing** that decomposes end-to-end provenance into distinct semantic layers (registry, specification, morphisms, plans, artifacts, score, selection, data), enabling per-layer fault localization without re-execution (§3.3).
4. **Description/identity separation** in a content-addressed capability registry, where discovery metadata is excluded from the capability hash, allowing the registry's natural-language interface to evolve independently of execution identity (§3.4).
5. **Empirical evaluation** of canonicalization collapse under 1,200 stochastic LLM generations, replay equivalence, provenance stability, and execution-graph divergence under stochastic generation (§4).

Scope and relationship to prior work. This paper focuses on execution semantics—what happens after a stochastic generator produces an expression or specification. It does not address generation quality, prompt engineering, or decoding strategies. A companion paper [7] established that stochastic proposals require an admissibility boundary (a grounding gate) before deterministic execution. The present paper investigates the execution abstractions on the deterministic side of that boundary.

We evaluate in one domain (systematic trading research) using one generator (Claude Sonnet). The architecture is domain-independent; the evaluation is not. We make no claims about financial performance—the domain serves as a deterministic execution substrate with computationally distinct, pairwise-confusable families. Runtime performance comparisons against Polars, Pandas, or other data-processing systems are out of scope; this paper is about execution semantics, not throughput.

2 Problem Statement

2.1 Terminology

We define terms precisely to avoid overloaded usage. All terms describe execution-level mechanisms, not cognitive or epistemic properties.

Definition 1 (Stochastic program generator)**.** *A system G that, given a natural-language prompt or specification p, produces an executable expression $e = G(p)$ non-deterministically. Independent invocations $G(p)_1, G(p)_2, \ldots$ may produce distinct expressions. This includes large language models, program synthesizers, evolutionary search, and any sampled-generation procedure.*

Definition 2 (Surface form)**.** *The syntactic representation of an expression as produced by a generator. Two surface forms are* distinct *if they differ as strings, regardless of whether they normalize to the same canonical form.*

Definition 3 (Canonical form)**.** *The unique normal form $\kappa(e)$ to which a surface-form expression e reduces under the normalization algebra. If $\kappa(e_1) = \kappa(e_2)$, then e_1 and e_2 are* canonically equivalent *—they produce the same execution plan, the same execution hash, and the same output given identical inputs.*

Definition 4 (Surface-form divergence)**.** *Given a computation target c (a description of the intended computation) and n independent generations $\{G(c)_1, \ldots, G(c)_n\}$, the surface-form divergence $\delta_S(c, n)$ is the number of distinct surface forms:*

$$\delta_S(c, n) = |\{G(c)_1, \ldots, G(c)_n\}|$$

Definition 5 (Canonical convergence)**.** *Given a set of expressions E and canonicalization function κ, the canonical convergence is:*

$$\gamma(E) = \frac{|E|}{|\{\kappa(e) : e \in E\}|}$$

Values greater than 1 indicate that canonicalization collapses surface-form divergence. A value of 1 means no collapse occurred.

Definition 6 (Morphism)**.** *A concrete executable configuration $m = (f, \theta, \kappa(e))$ consisting of a computational family f, a fully-bound parameter vector θ, and the canonical expression $\kappa(e)$ obtained by substituting θ into f's template and canonicalizing the result.*

Definition 7 (Typed specification)**.** *A declarative tuple $s = (family, metric, param\_ranges, source)$ from which morphisms are generated by grid expansion. The generator proposes s; the execution system generates morphisms. The generator never writes the executable expression directly.*

Definition 8 (Execution hash)**.** *A content-addressed fingerprint $H(x) = \text{SHA-256}(h_1 \parallel \cdots \parallel h_k)$ computed over k semantic sub-hashes, each covering a distinct layer of the computation. In this paper, $k = 8$.*

Definition 9 (Replay equivalence)**.** *Two executions x_1, x_2 of specification s on data d with registry R are* replay-equivalent *iff $H(x_1) = H(x_2)$. Replay equivalence is verified by hash comparison, without re-execution.*

Definition 10 (Capability entry)**.** *A registry record $c = (SEM, ALG, IMPL, DISC)$ with identity hash $h_c = \text{SHA-256}(SEM \parallel ALG \parallel IMPL)$. The discovery layer $DISC$ is excluded from h_c.*

Definition 11 (Description/identity separation)**.** *For capability c, the identity hash h_c is invariant under changes to $DISC(c)$. Formally: for any $DISC'$, $h_{(c.SEM,\, c.ALG,\, c.IMPL,\, DISC')} = h_c$. Improving how a capability is described does not change what it computes.*

2.2 Execution Problems Under Stochastic Generation

We identify five execution-level problems that arise when the primary generator of computation is stochastic. These are not generation failures (hallucination, malformed output, wrong capability selection); those are addressed by admission mechanisms such as schema validation and grounding gates [7]. The problems below arise even when the generator produces correct, well-formed, admitted expressions.

P1: Surface-form divergence. Independent runs of a stochastic generator produce syntactically different expressions that normalize to the same canonical form. In human-authored code, this variation is cosmetic (caught by a formatter, ignored by a reviewer). In an execution system that content-addresses results, it produces different hashes for expressions that normalize to the same canonical form, defeating deduplication, caching, and replay comparison.

Example. A generator asked for "log returns" may produce `(dlog x)`, `(ln_diff x)`, `(-> x (dlog))`, or `(-> x (ln) (diff 1))` across four runs. Without normalization, the system treats these as four distinct computations.

P2: Provenance fragmentation. Without canonical forms, pipelines that normalize to the same executable representation generate different provenance records. A researcher who ran a pipeline last month cannot verify it against a pipeline generated today if the generator produced a different surface form. The provenance chains are structurally different even though both normalize to the same canonical form.

Consequence. Replay verification degrades from hash comparison (constant time, no re-execution) to output comparison (requires re-execution and tolerance-based matching).

P3: Morphism explosion. When specifications are generated stochastically, surface-form variation in parameter expressions or template instantiations produces duplicate morphisms that differ only in surface form. Without canonical deduplication, the system evaluates redundant configurations, wasting computation and inflating result sets.

P4: Registry-evolution coupling. A capability registry must evolve: new aliases are added, descriptions are improved, tags are updated. In content-addressed systems that hash all metadata (Nix, Guix), these changes alter the capability hash and cascade to every downstream execution hash. Prior provenance records become invalid even though the computation is unchanged.

Consequence. Registry maintenance invalidates provenance, creating a tension between discoverability (improving descriptions for stochastic generators) and reproducibility (preserving execution hashes for verification).

P5: Divergence localization. When two execution hashes differ, the system must determine *where* they diverge. A monolithic hash provides a single bit of information (match/mismatch). Without decomposition, diagnosing whether the divergence originates in the registry, the specification, the data, the execution plan, or the scoring requires re-execution and manual comparison.

2.3 Formal Problem

Given a stochastic program generator G, a computation target c, and n independent generations, the execution system must provide:

1. **Canonical collapse.** A normalization function κ such that the canonical convergence $\gamma > 1$ for non-trivial n. That is, κ must collapse at least some surface-form divergence: $|\{\kappa(G(c)_i)\}| < |\{G(c)_i\}|$ when the generator produces distinct surface forms that normalize to the same canonical form.
2. **Replay determinism.** For all specifications s, data d, and registry states R: executing s on d with R twice produces replay-equivalent executions. Formally, $H(x_1) = H(x_2)$ where x_1 and x_2 are independent executions of the same triple (s, d, R).
3. **Provenance stability.** Registry changes restricted to discovery metadata (DISC) must not alter any execution hash. Formally, for registry change $R \to R'$ where only DISC fields differ: $H_R(x) = H_{R'}(x)$.
4. **Divergence decomposition.** When $H(x_1) \neq H(x_2)$, the system must identify the semantic layer at which the hashes first diverge, without re-executing either x_1 or x_2.
5. **Morphism deduplication.** The morphism expansion from specification to executable configurations must identify and eliminate duplicates by canonical hash, so that surface-form variation in the specification does not produce redundant execution.

These requirements are execution-level: they concern the system's behavior after an expression or specification has been generated and admitted. They are orthogonal to generation quality and admission control, which operate upstream.

2.4 Scope and Non-Claims

What this paper addresses. BLISP addresses six execution-level problems caused by stochastic program generation: surface-form divergence (P1), provenance fragmentation (P2), morphism duplication (P3), registry-evolution coupling (P4), divergence localization (P5), and replay instability (the general case of P1–P5 compounding across pipeline stages).

What this paper does not address. BLISP does not address semantic correctness, scientific validity, optimality of generated programs, natural-language understanding, or adversarial intent inference. The system does not determine whether a generated program is the *right* program for the user's goal—only whether it is canonically well-formed, content-addressable, and replayable. Admission control (whether a proposal should be allowed to execute) is handled upstream by grounding gates [7] and is outside the scope of this paper.

Canonical equivalence is system-relative. Two expressions are equivalent for the purposes of this paper if and only if they normalize to the same BLISP canonical form and therefore produce the same execution plan, the same execution hash, and the same output under identical inputs and registry state. We do not claim universal semantic equivalence. The normalization algebra defines a specific set of rewrite rules (threading, aliasing, argument reordering); expressions that are equivalent under a richer equational theory but not under these rules will not be identified as equivalent by the system. The paper measures canonicalization collapse within this algebra, not equivalence under an external semantics.

2.5 Execution Pipeline Model

We model the end-to-end pipeline from natural-language prompt to execution hash as a composition of five stages. This model is operational, not denotational: it describes the pipeline structure that the architecture (§3) instantiates and the evaluation (§4) measures.

Definition 12 (Execution pipeline)**.** *Given a prompt p, the execution pipeline is:*

$$H \circ E \circ \kappa \circ \Gamma \circ G$$

where:

- $G : \mathcal{P} \to \mathcal{S}$ *is the* ***stochastic generator****. Given prompt $p \in \mathcal{P}$, it produces a typed specification $s = G(p)$. We abuse notation slightly by writing $G : \mathcal{P} \to \mathcal{S}$; operationally, G is stochastic and independent invocations $G(p)_1, G(p)_2, \ldots$ may produce distinct specifications.*
- $\Gamma : \mathcal{S} \times \mathcal{R} \to \mathcal{S}_\checkmark$ *is the* ***grounding gate****. It admits or rejects a specification based on registry evidence. Γ is deterministic given a registry state R: $\Gamma(s, R)$ either returns the admitted specification or rejects.*
- $\kappa : \mathcal{S}_\checkmark \to \mathcal{M}^*$ *is* ***canonicalization and expansion****. It expands an admitted specification into a set of canonical morphisms via grid expansion and normalization. κ is deterministic: the same specification and registry always produce the same morphism set.*
- $E : \mathcal{M}^* \times \mathcal{D} \to \mathcal{A}$ *is* ***execution****. It evaluates morphisms against data $d \in \mathcal{D}$, producing artifacts (plans, results, scores). E is deterministic: same morphisms and data produce identical artifacts.*
- $H : \mathcal{A} \to \{0,1\}^{256}$ *is the* ***execution hash****. It computes a content-addressed fingerprint from the 8-layer decomposition of artifacts. H is deterministic.*

The pipeline has exactly one non-deterministic stage: G. All stages after G—grounding, canonicalization, execution, hashing—are deterministic functions of their inputs. The *canonical execution boundary* is the interface between G (stochastic) and $\kappa \circ \Gamma$ (deterministic). After this boundary:

$$\kappa(s_1) = \kappa(s_2) \;\Rightarrow\; H(E(\kappa(s_1), d)) = H(E(\kappa(s_2), d))$$

for any data d and registry state R. Throughout this paper, we suppress the registry argument in equations: $E(\kappa(s), d)$ is shorthand for $E(\kappa(s), d, R)$ with R held fixed.

Three properties hold for the deterministic portion of the pipeline:

1. **Determinism.** κ, E, and H are total deterministic functions. Given identical inputs, they produce identical outputs.
2. **Idempotence.** $\kappa(\kappa(s)) = \kappa(s)$. Canonicalizing an already-canonical specification is a no-op.
3. **Boundary.** For all s_1, s_2: if $\kappa(s_1) = \kappa(s_2)$, then $H(E(\kappa(s_1), d)) = H(E(\kappa(s_2), d))$ for all d. Surface-form variation in specifications does not propagate past canonicalization.

What this model does not claim. The pipeline model describes operational structure. It does not provide a denotational semantics, does not claim confluence or completeness of κ, and does not address whether G produces the "correct" specification for a given prompt. The model's purpose is to make the boundary property precise: exactly one stage is stochastic, and its non-determinism does not propagate.

3 The Canonical Execution Boundary

The architecture enforces a single invariant: the *canonical execution boundary*. The pipeline is partitioned into a stochastic upstream—the generator, admission control, surface-form variation—and a deterministic downstream—canonicalization, planning, execution, provenance. Stochasticity does not propagate beyond this boundary.

Four mechanisms enforce the boundary. Typed specifications constrain what the generator produces (§3.1). The canonicalization pipeline normalizes what remains (§3.2). Content-addressed execution hashing tracks the result (§3.3). Description/identity separation insulates execution identity from registry evolution (§3.4). We describe each, then state the formal properties (§3.5).

3.1 Typed Specification as Constrained Stochastic Proposal

The typed specification is the paper's central architectural response to stochastic generation. Rather than allowing the generator to produce arbitrary executable expressions—which would require canonicalization to collapse surface-form variation after the fact—the architecture constrains the generator to produce structured proposals from a finite semantic space. The execution system, not the generator, produces the executable expression.

The stochastic generator proposes a *typed specification*: a declarative tuple naming a computational family, a scoring metric, parameter ranges, and a data source. The execution system expands this specification into concrete executable configurations (morphisms) via Cartesian product over the parameter ranges.

Specification structure. A specification contains:

- **Family**: a registered computational family (e.g., `MOM_WZS`, `CARRY`) with a typed parameter schema and expression templates.
- **Metric**: the scoring function used to rank morphisms (e.g., `SRP` for Sharpe ratio, `MDD` for maximum drawdown).
- **Parameter ranges**: for each parameter in the family's schema, either a single value or a list of values to sweep.
- **Source**: the data file against which morphisms are executed.

Family registry. Each family is registered with a typed schema: a list of named parameters with types, defaults, and descriptions; a signal expression template; and a risk expression template. For example, `MOM_WZS` (z-scored momentum) has 7 parameters (`SIG_WIN`, `SIG_STP`, `RSK_WIN`, `RSK_STP`, `TGT_VOL`, `LAG`, `LVG`) with defaults and two expression templates containing substitution placeholders. The registry currently contains 4 families; the architecture imposes no limit.

Grid expansion. The system takes the Cartesian product of all parameter ranges, producing one morphism per combination. For each morphism:

1. Substitute bound parameters into the family's expression templates.
2. Canonicalize the resulting expression through the normalization pipeline (§3.2).
3. Compute the content-addressed hash of the canonical expression.

Deduplication. After expansion, morphisms with identical canonical hashes are collapsed. This addresses P3 (morphism duplication): if two parameter combinations produce the same canonical expression (e.g., because a parameter has no effect on the expression for certain values), the system executes the expression once.

Architectural consequence. The generator operates at the specification level: it selects a family and parameter ranges. The execution system controls the expansion from specification to executable morphisms. This separation means that surface-form divergence in expression generation (P1) is eliminated by construction—the generator never writes the expression. Divergence can still occur at the specification level (different family or parameter choices), but this is *intent* divergence, not surface-form divergence. In terms of the pipeline model (§2.5), the typed specification constrains G to produce elements of a structured space $\mathcal{S}$; canonicalization κ then expands and normalizes these into deterministic morphism sets.

3.2 Canonicalization Pipeline

Every expression passes through a four-stage pipeline before execution: **parse** → **normalize** → **canonicalize** → **plan**. The first three stages are deterministic transformations that reduce an arbitrary surface form to a unique canonical representation. The fourth stage compiles the canonical form into an intermediate representation (IR) for execution. Each stage is traced: the system records which rewrites were applied, enabling inspection of how a surface form reached its canonical form.

Stage 1: Parse. The parser accepts S-expressions and produces an abstract syntax tree (AST). No normalization occurs. The surface form is preserved exactly as the generator produced it.

Stage 2: Normalize. The normalizer applies three rewrite classes, in order:

(a) Thread-first expansion. The threading macro `->` is a surface-level convenience that passes a value through a chain of operations:

```
(-> x (f a) (g b)) ⟶ (g (f x a) b)
```

The value is inserted as the first argument of each subsequent form. This eliminates pipeline-style variation: the threaded and nested forms normalize to the same expression. Without this rewrite, a stochastic generator choosing between pipeline and nested style produces different hashes for expressions that produce identical execution plans.

(b) Alias resolution. The system maintains 51 alias rules that map alternative names to canonical names. Examples:

Alias	Canonical
`cs1`, `cumsum`	`run_sum`
`wzs`	`rol_zsc`
`w5`	`wkd`
`add`	`+`
`ln`	`log`
`div`	`/`

Each alias maps to exactly one canonical name. A stochastic generator that produces `wzs` in one run and `rol_zsc` in another will yield the same canonical form after alias resolution. In the static registry, 278 accepted surface forms collapse to 235 canonical operations through alias resolution, eliminating 43 naming variants.

(c) Argument reordering. For binary operations where one argument is a data operand and the other is a numeric parameter, the normalizer enforces a canonical argument order (data-first). For example:

```
(rol_avg 250 x) ⟶ (rol_avg x 250)
```

This eliminates argument-order variation: a generator that places the window parameter before or after the data operand produces the same canonical form. The rewrite is applied only when the argument types are unambiguous (one integer, one non-integer), preventing incorrect reordering of two-data-argument operations.

Stage 3: Canonicalize. The canonical form is the deterministic string serialization of the normalized AST. This string is the input to the identity hash: two expressions with the same canonical string produce the same SHA-256 hash, the same execution plan, and—given identical inputs and registry—the same output.

Stage 4: Plan. The planner compiles the canonical form into an intermediate representation (IR) consisting of typed operation nodes. The IR has 66 operation variants across 8 categories (numeric, binary, join, schema, aggregate, matrix, ternary, scan). Planning is deterministic: the same canonical form always produces the same IR plan.

Design rationale. Traditional compilers normalize to optimize: constant folding, dead-code elimination, and strength reduction transform programs into faster but not necessarily unique forms. Multiple valid intermediate representations may exist. BLISP normalizes for *execution identity*: the goal is a single canonical form per computation, so that content-addressed hashing, deduplication, and replay comparison operate correctly under stochastic surface-form variation. Optimization (IR fusion) is a separate, subsequent pass that does not alter the canonical identity.

Quantitative summary. Table 1 summarizes the static canonicalization profile of the BLISP registry.

Table 1: Static canonicalization profile. The registry accepts 278 surface-form names that collapse to 235 canonical operations.

Metric	Value
Accepted surface forms	278
Canonical operations	235
Aliases collapsed	43
Static collapse ratio	1.18
Rewrite classes	3
Alias rules	51
IR operation variants	66
IR operation categories	8

3.3 Content-Addressed Execution Hashing

Every completed execution produces an 8-layer content-addressed hash. Each sub-hash covers a distinct semantic layer of the computation, enabling both replay verification and divergence localization.

Table 2: Execution hash decomposition. Each sub-hash covers a distinct computation layer. When two execution hashes differ, comparing sub-hashes localizes the divergence without re-execution.

#	Sub-hash	Covers	Sensitive to
1	`DIC_HSH`	All capability hashes (sorted)	Registry changes
2	`FPR_HSH`	Family, metric, parameters	Specification changes
3	`MOR_HSHS`	All morphism expressions (canonical)	Grid expansion changes
4	`SRH_HSHS`	Compiled execution plans	Planner changes
5	`CMPN_HSHS`	Output data artifacts	Execution result changes
6	`SCR_HSH`	Scoring result (metric values)	Scoring changes
7	`SEL_HSH`	Best-candidate identity	Selection changes
8	`DATA_HSH`	Source data content	Input data changes

The eight layers. The top-level execution hash is $H = \text{SHA-256}(h_1 \parallel h_2 \parallel \cdots \parallel h_8)$, where each h_i is the sub-hash for layer i.

Replay verification. Given two executions of the same specification on the same data with the same registry, replay equivalence holds iff $H(x_1) = H(x_2)$. This is verified by hash comparison alone—no re-execution required. The determinism guarantee is architectural: every stage in the pipeline (parsing, normalization,

planning, execution, scoring, selection) is deterministic, and all inputs are content-addressed. Non-determinism in the generator does not propagate past the canonicalization boundary.

Divergence localization. When $H(x_1) \neq H(x_2)$, comparing sub-hashes identifies the divergence layer in $O(k)$ comparisons (where $k = 8$). Examples:

- h_1 differs $\Rightarrow$ the capability registry changed (an operation was added, removed, or modified).
- h_1 matches but h_2 differs $\Rightarrow$ the specification changed (different family, metric, or parameters).
- h_1–h_6 match but h_7 differs $\Rightarrow$ the best candidate changed (a different morphism won under the same scoring), likely due to data or scoring-threshold changes.
- h_8 differs $\Rightarrow$ the source data changed.

This addresses P5 (divergence localization): the system provides per-layer diagnosis without re-execution.

3.4 Capability Identity

Each capability in the registry is a 4-layer tuple. Three layers determine the capability's identity hash; the fourth is excluded.

Identity layers.

- **Semantic (SEM)**: algebraic properties—pure, deterministic, commutative, associative, idempotent, orientation, stateful.
- **Algebraic (ALG)**: type signature—input types, output types, arity, reversibility.
- **Implementation (IMPL)**: execution details—engine, IR variant, fusability, execution path.

The identity hash is:

$$h_c = \text{SHA-256}(\text{SEM}(c) \parallel \text{ALG}(c) \parallel \text{IMPL}(c))$$

Discovery layer (excluded). The fourth layer, **Discovery (DISC)**, contains aliases, descriptions, tags, usage examples, and deprecation status. DISC is intentionally excluded from h_c.

The exclusion resolves P4 (registry-evolution coupling). A capability registry that serves stochastic generators must evolve its natural-language interface: adding aliases so that generators discover capabilities under more terms, improving descriptions so that generators select capabilities more accurately, updating tags for categorization. In a system that hashes all metadata (Nix, Guix), each of these changes invalidates prior execution hashes. With description/identity separation, these changes affect what generators can discover but not what capabilities compute and not any prior execution hash.

Registry hash. The full registry hash h_R is the SHA-256 of the sorted concatenation of all capability identity hashes:

$$h_R = \text{SHA-256}(\text{sort}(\{h_c : c \in R\}))$$

This produces a single fingerprint of the capability algebra. Every execution hash includes h_R (as `DIC_HSH`), making the entire provenance chain sensitive to changes in the set of available operations.

Consequence. Adding an alias like "log returns" $\rightarrow$ `dlog` changes DISC(`dlog`), which is excluded from $h_{\texttt{dlog}}$. The registry hash h_R is unchanged. All downstream execution hashes remain valid. A researcher who executed a pipeline six months ago can verify their execution hash against the current registry even if the registry's discovery metadata has been extensively updated in the interim.

Conversely, changing the type signature of `dlog` (an ALG change) alters $h_{\texttt{dlog}}$, which alters h_R, which cascades to every execution hash that includes `DIC_HSH`. This is the correct behavior: the computation changed, so the provenance record should reflect it.

Quantitative summary. The registry contains 236 capability entries across three execution layers: 116 with IR execution paths, 148 with legacy execution paths, and 14 in the language-construct (GLUE) layer. Each entry has all four layers populated. The registry hash is recomputed on startup from the live capability set.

3.5 Properties of the Normalization Algebra

The normalization algebra implements three rewrite classes (§3.2). We state three operational properties that can be verified by inspection of the rewrite rules. We do not claim confluence or completeness—these would require a formal equational theory that the system does not provide.

Property 1 (Determinism)**.** *For any surface-form expression e, the canonicalization function κ produces a unique output: $\kappa(e)$ is a total, deterministic function from expressions to canonical forms. There are no non-deterministic choices in the rewrite rules.*

Justification. Each rewrite class applies a fixed set of rules in a fixed order. Thread-first expansion is a structural transformation on the AST (the `->` macro is expanded left-to-right). Alias resolution consults a static table (each alias maps to exactly one canonical name). Argument reordering applies when the argument types are unambiguous (one integer, one non-integer) and enforces a single canonical order. No rule consults randomness, external state, or input data. The pipeline stages execute in a fixed sequence: parse, normalize, canonicalize, plan.

Property 2 (Idempotence)**.** *Canonicalization is idempotent: $\kappa(\kappa(e)) = \kappa(e)$ for all expressions e. Applying the normalization pipeline to an already-canonical expression produces the same expression.*

Justification. Thread-first expansion has no effect on expressions without the `->` macro (canonical forms never contain `->`). Alias resolution has no effect when all names are already canonical (no alias maps a canonical name to a different canonical name). Argument reordering has no effect when arguments are already in canonical order. Since each rewrite class is a no-op on canonical forms, the composition is idempotent.

Property 3 (Finite termination)**.** *For any expression e, $\kappa(e)$ terminates in time bounded by the product of the number of rewrite rules and the size of the expression.*

Justification. Thread-first expansion runs a single pass over the AST, eliminating one `->` node per step. The expression strictly shrinks in `->` count at each step; since the count is finite, the pass terminates. Alias resolution makes a single pass over the AST, replacing each name at most once (no alias chain cycles exist: each alias maps to a canonical name that is not itself an alias). Argument reordering makes a single pass, examining each binary application once. The total work is $O(|R| \cdot |e|)$, where $|R|$ is the number of rewrite rules and $|e|$ is the expression size.

What we do not claim. The normalization algebra is not confluent in the general sense: two structurally different expressions that produce the same output under all inputs will not necessarily reduce to the same canonical form. For example, `(+ x x)` and `(* x 2)` produce identical outputs for numeric x but are distinct canonical forms. Canonicalization collapses *surface-form* variation (naming, threading, argument order); it does not identify all extensionally equivalent programs. The evaluation (§4) measures the collapse that the algebra achieves within its rewrite classes, not equivalence under a richer equational theory.

3.6 Canonical Execution Boundary

The properties above yield the paper's central architectural invariant. The formal pipeline model (§2.5) makes this boundary precise: G is the only non-deterministic stage; κ, E, and H are deterministic.

Property 4 (Canonical execution boundary)**.** *After canonicalization, all downstream execution artifacts—the IR plan, execution results, scoring, selection, and provenance hashes—are deterministic functions of three inputs: the canonical form $\kappa(e)$, the registry state R, and the input data d.*

Formally. Let F denote the composition of all post-canonicalization stages (planning, execution, scoring, selection, hashing), with the registry state R made explicit. Then:

$$F(\kappa(e_1), R, d) = F(\kappa(e_2), R, d) \quad \text{whenever} \quad \kappa(e_1) = \kappa(e_2)$$

The boundary partitions the pipeline into a stochastic upstream (the generator, surface-form variation, admission control) and a deterministic downstream (planning, execution, provenance). The key invariant is:

Stochasticity does not propagate beyond the canonical execution boundary.

This invariant is architectural, not probabilistic. No component downstream of canonicalization consults randomness, the generator's state, or any non-deterministic input. The IR planner is a deterministic function of the canonical AST and the registry. The executor is a deterministic function of the IR plan and the input data. The hasher is a deterministic function of the execution artifacts. If two expressions produce the same canonical form, they produce the same execution hash, regardless of which generator produced them, how many times they were generated, or what surface form they arrived in. This is the paper's central invariant. Every mechanism described in §3.1 through §3.5 exists to enforce it. The evaluation (§4) measures whether it holds empirically.

Relationship to the grounding gate. The grounding gate [7] is the admission boundary between stochastic proposals and deterministic execution. The canonical execution boundary is the *identity* boundary: the point at which surface-form divergence is resolved and execution identity is fixed. The two boundaries are complementary. The grounding gate ensures that only admissible proposals reach the execution system. The canonical execution boundary ensures that admitted proposals with different surface forms but the same canonical form produce identical downstream artifacts.

Consequence for replay. Replay equivalence (Definition 9) follows directly from the boundary property. If $\kappa(e_1) = \kappa(e_2)$ and $R_1 = R_2$ and $d_1 = d_2$, then $H(x_1) = H(x_2)$. Replay verification reduces to comparing canonical forms (or, equivalently, comparing any sub-hash that covers the canonical form). No re-execution is needed.

4 Evaluation

The central empirical thesis is: *canonicalization measurably reduces stochastic execution divergence.* Every experiment in this section tests a specific aspect of that thesis. We measure canonicalization collapse (§4.2), replay equivalence (§4.3), provenance stability (§4.4), morphism deduplication (§4.5), and specification-level versus expression-level divergence (§4.6).

4.1 Experimental Setup

System under test. BLISP: 60,888 lines of Rust (source), 34,130 lines of tests, 1,663 test cases. The canonicalization pipeline, morphism expansion, and content-addressed hashing operate as described in §3. All experiments run against the same codebase and registry snapshot. Experiment scripts, expected outputs, and registry snapshots are in the artifact repository (`artifacts/canonical-execution/`; see `reproducibility/README.md` for verification instructions).

Registry state. The capability registry contains 278 accepted surface forms, 235 canonical operations, 51 alias rules across 3 rewrite classes, 66 IR operation variants, and 4 registered computational families (`MOM_WZS`, `CARRY`, `MOM_REV`, `VOL_TGT`).

Data. Experiments use canonical data files: `At.csv` (multi-asset, ∼500 columns, ∼5100 rows) and `smallAt.csv` (6 columns, 9 rows). No ad-hoc data is generated.

Hash verification. All execution hashes use SHA-256. Replay equivalence is verified by exact hash comparison (no tolerance). Sub-hash comparison uses the 8-layer decomposition described in §3.3.

Measurement methodology. Static registry measurements are extracted from the live capability dictionary (`blisp -dic`). Dynamic measurements (morphism expansion, hash stability, execution identity) are performed by constructing typed specifications, expanding them through the pipeline, and comparing content-addressed hashes across runs. Each dynamic experiment is executed at least twice to verify determinism.

4.2 Canonicalization Collapse

This experiment measures whether G's stochastic variation propagates past κ (§2.5)—i.e., how effectively canonicalization collapses surface-form divergence.

Static registry collapse. The capability dictionary provides a complete enumeration: every accepted name maps to exactly one canonical name. Table 3 shows the collapse by rewrite class.

Table 3: Surface-form collapse by rewrite class. The 43 collapsed aliases fall into three categories. Some surface forms are affected by multiple rewrite classes (e.g., a threaded alias expression undergoes both threading and alias resolution).

Rewrite class	Aliases	Example
Alias resolution	43	`wzs` → `rol_zsc`, `cs1` → `run_sum`, `w5` → `wkd`
Thread-first expansion	(structural)	`(-> x (f) (g))` → `(g (f x))`
Argument reordering	(structural)	`(rol_avg 250 x)` → `(rol_avg x 250)`

Alias resolution collapses 43 distinct surface-form names to their canonical targets. Threading and argument reordering are structural rewrites that cannot be enumerated statically (they depend on expression shape), but apply uniformly to all expressions passing through the pipeline. The static collapse ratio is $278/235 = 1.18$.

Dynamic collapse under expression variation. To demonstrate that canonicalization collapses variation in practice, consider the computation "rolling z-score of log returns." The following four expressions, which a stochastic generator might produce across independent runs, all differ as strings:

#	Surface form
1	`(-> (stdin) (dlog) (wzs 25 1))`
2	`(-> (stdin) (dlog) (rol_zsc 25 1))`
3	`(rol_zsc (dlog (stdin)) 25 1)`
4	`(wzs (dlog (stdin)) 25 1)`

Expression 1 uses threading (`->`) and the alias `wzs`. Expression 2 uses threading with the canonical name `rol_zsc`. Expression 3 uses nested form with the canonical name. Expression 4 uses nested form with the alias. After canonicalization, all four reduce to the same canonical form:

```
(rol_zsc (dlog (stdin)) 25 1)
```

This was verified by executing all four against identical data and comparing output checksums: all four produce byte-identical output. The canonical convergence for this example is $\gamma = 4/1 = 4$: four distinct surface forms collapse to one canonical form.

Collapse by alias category. The 43 collapsed aliases fall into recognizable categories:

- **Mathematical shorthands** (9): `add` → `+`, `sub` → `-`, `mul` → `*`, `div` → `/`, `eq` → `==`, `neq` → `!=`, `gt` → `>`, `lt` → `<`, `ln` → `log`.
- **Columnar variants** (16): `dlog-col`/`dlog-cols` → `dlog`, `shift-col`/`shift-cols` → `shift`, and similar `-col`/`-cols` suffixed aliases for 8 operations.
- **Domain aliases** (7): `wzs` → `rol_zsc`, `cs1` → `run_sum`, `ur` → `rsk_adj`, `w5` → `wkd`, `nmax` → `rol_max`, `nmin` → `rol_min`, `x-` → `xminus`.
- **Other** (11): comparison aliases (`gte` → `>=`, `lte` → `<=`), composite columnar aliases (`ur-col`/`ur-cols` → `rsk_adj`), `let*` → `let`.

Each category represents a distinct source of stochastic surface-form divergence. A generator trained on mathematical notation may produce `add`; one trained on programming languages may produce `+`. Both reduce to the same canonical form.

Stochastic generation experiment. To measure canonicalization collapse under actual stochastic generation, we ran 1,200 independent LLM calls (Claude Sonnet, via the typed specification pipeline): 30 prompts $\times$ 4 temperatures ($T \in \{0.0, 0.3, 0.7, 1.0\}$) $\times$ 10 repetitions. The 30 prompts span all 4 registered families and 5 difficulty categories (straightforward, confusable, adversarial, undiscoverable, multi-family). All 1,200 calls produced valid specifications (100% validation rate).

For each generation, we recorded the raw FPR JSON, computed the content-addressed specification hash (`FPR_HSH`) via the BLISP pipeline, and measured distinct surface specifications (δ_S), distinct canonical specifications (δ_C), and the collapse ratio $\gamma = \delta_S/\delta_C$ per (prompt, temperature).

Table 4 summarizes the aggregate results.

Table 4: Specification stability under stochastic generation. 30 prompts $\times$ 10 reps at each temperature. $P(\delta_F{=}1)$: fraction of prompts where all 10 runs agree on family.

T	Med. δ_S	Med. δ_C	$P(\delta_C{=}1)$	$P(\delta_F{=}1)$	n_v/n
0.0	1	1	1.00	1.00	300/300
0.3	1	1	0.93	0.93	300/300
0.7	1	1	0.83	0.90	300/300
1.0	1	1	0.87	0.93	300/300

At T=0.0, all 30 prompts produce identical specifications across all 10 runs (δ_C=1, δ_F=1). At T=0.7, 83% of prompts still produce a single canonical specification, and 90% agree on the family. The prompts that diverge (3 of 30 at T=0.7) do so at the specification level—different family or different source path—not at the surface-form level. The canonical hash correctly treats these as distinct specifications.

Interpretation. The specification-level pipeline produces $\gamma \approx 1$ because surface-form divergence is eliminated by construction: the generator proposes typed specifications, not executable expressions. The remaining divergence is *intent* divergence (different family or metric selections), which canonicalization correctly does not collapse. The static collapse ratio of 1.18 measures the canonicalization algebra's capacity; the stochastic experiment measures the combined architecture's effectiveness. Together they show that the canonicalization layer provides a guarantee (any surface variation is collapsed) while the specification layer prevents the variation from occurring in the first place. This result is a direct consequence of the canonical execution boundary: by constraining the generator to produce typed specifications rather than executable expressions, surface-form divergence is eliminated before the boundary is reached.

4.3 Replay Equivalence

This experiment verifies the boundary property (§2.5): that identical specifications executed against identical data and registry state produce identical execution hashes—i.e., $\kappa(s_1) = \kappa(s_2) \Rightarrow H(x_1) = H(x_2)$.

Method. We construct five typed specifications spanning all four registered families: `MOM_WZS` (3-param sweep), `CARRY` (defaults), `MOM_REV` (3-param sweep), `VOL_TGT` (3-param sweep), and `MOM_WZS` (2-param sweep, 3×2 grid). Each specification is expanded through the morphism pipeline, and the content-addressed hash (`FPR_HSH`) and morphism count are recorded. Each specification is executed 10 times independently.

Result. Table 5 summarizes the results. All 50 runs produce bit-identical hashes within each specification. Morphism counts are deterministic: 3, 1, 3, 3, and 6 respectively.

Table 5: Replay equivalence: 5 specifications $\times$ 10 independent runs. All hashes are identical within each specification; all specifications produce distinct hashes.

Family	Parameters	MOR	`FPR_HSH` (prefix)	Identical
`MOM_WZS`	`SIG_WIN`$\in \{25, 50, 100\}$	3	`b4ad4d...`	10/10
`CARRY`	(defaults)	1	`675595...`	10/10
`MOM_REV`	`SIG_WIN`$\in \{25, 50, 100\}$	3	`5eed78...`	10/10
`VOL_TGT`	`SIG_WIN`$\in \{20, 40, 80\}$	3	`1b113d...`	10/10
`MOM_WZS`	3×2 grid	6	`d2822e...`	10/10

All five specifications produce distinct `FPR_HSH` values (5 of 5 unique), confirming that the hash is collision-free across these inputs. The specification hash is a deterministic function of the specification content, not of run order, timestamp, or process ID.

Sub-hash decomposition. Because each layer of the execution hash is deterministic and content-addressed, replay equivalence holds at every layer independently. If two executions of the same specification on the same data produce different top-level hashes, sub-hash comparison identifies the divergent layer in $O(k)$ comparisons ($k = 8$), without re-execution. This is the mechanism for P5 (divergence localization).

Consequence. Replay verification is a hash comparison, not an output comparison. It requires no re-execution, no tolerance thresholds, and no statistical tests. A researcher can verify that a pipeline executed today has identical execution identity to one executed six months ago by comparing the 8-layer hash alone.

4.4 Provenance Stability Under Registry Evolution

This experiment tests whether changes restricted to the grounding gate's discovery metadata (Γ's DISC layer) alter the execution hash H—i.e., whether discovery-metadata changes to the capability registry invalidate prior execution hashes.

Design. The capability registry separates identity (SEM, ALG, IMPL) from discovery (DISC). The registry hash h_R is the SHA-256 of the sorted concatenation of all capability identity hashes (§3.4). The experiment has two conditions:

1. **DISC change**: add an alias to the DISC layer of an existing capability (e.g., adding "log returns" as a discovery alias for `dlog`). Measure whether h_R changes.
2. **SEM change** (control): modify a semantic property of an existing capability (e.g., changing the "pure" flag). Measure whether h_R changes.

Result. We run `AGENT-DISCOVER` with 10 distinct term sets (e.g., {"momentum", "sharpe"}, {"carry", "drawdown"}, {"log returns", "momentum"}). Each invocation exercises different DISC-layer aliases and matching paths, yet all 10 return the identical `DIC_HSH`: `f108b3...`. Repeating the same term set 10 times also produces an identical hash. The registry hash is a function of the SEM/ALG/IMPL layers, not of which discovery queries have been issued against it.

Under SEM changes (e.g., modifying the "pure" flag of an operation), h_c changes by construction, which cascades to `DIC_HSH` and all downstream execution hashes. This is the correct behavior: the computation changed, so the provenance record reflects it.

Quantitative scope. The registry contains 236 capability entries. Each entry has all four layers populated. The DISC layer contains aliases (consumed by `AGENT-DISCOVER` for stochastic generators), descriptions, tags, and deprecation status. These fields evolve frequently as the system's natural-language interface improves. Over the development history, DISC changes outnumber identity changes by approximately an order of magnitude. Without description/identity separation, each DISC change would invalidate all prior execution hashes.

Consequence. Description/identity separation resolves the tension between discoverability and reproducibility (P4). The registry can improve how stochastic generators discover capabilities—adding aliases, improving descriptions, updating tags—without breaking any prior provenance chain. This is not possible in content-addressed systems that hash all metadata (Nix, Guix), where adding a maintainer field or updating a description invalidates the derivation hash.

4.5 Morphism Deduplication

This experiment measures whether typed specification and canonical hashing eliminate redundant morphisms.

Grid expansion. Morphism expansion takes the Cartesian product of parameter ranges. Table 6 shows the relationship between parameter ranges and morphism counts for the `MOM_WZS` family.

Table 6: Grid expansion: morphism count equals the Cartesian product of parameter range sizes. The expansion is exact and deterministic.

Parameter ranges	Grid	Morphisms
`SIG_WIN` $\in \{25, 50, 100\}$	3	3
`SIG_WIN` $\in \{25, 50, 100\}$ $\times$ `RSK_WIN` $\in \{125, 250\}$	3×2	6

Each morphism is a concrete executable configuration: the family's expression templates are instantiated with bound parameters, canonicalized, and content-addressed. The morphism hash is a deterministic function of the canonical signal expression, canonical risk expression, bound parameters, and the registry hash.

Content-addressed deduplication. After expansion, morphisms with identical content-addressed hashes are candidates for deduplication. Two parameter combinations that produce the same canonical expression (because a parameter has no effect for certain values) would hash identically and be collapsed. The deduplication function (`dedup_mors`) operates by inserting morphism hashes into a set and retaining only unique entries.

Consequence. The typed-specification architecture eliminates P3 (morphism explosion) by construction: the generator proposes parameter ranges, not expressions. Expansion is a deterministic Cartesian product. Duplicate parameter values produce duplicate morphisms that are identified by hash. A stochastic generator that proposes overlapping parameter ranges across independent runs does not produce redundant execution—the content-addressed hashing identifies the duplicates.

4.6 Specification-Level vs. Expression-Level Generation

This experiment compares divergence properties at two levels of abstraction: stochastic generators proposing typed specifications versus directly generating executable expressions.

Expression-level generation. When a generator produces executable expressions directly, surface-form divergence propagates to every downstream artifact. Consider n independent generations targeting the same canonical form. Without canonicalization, the system sees up to n distinct expressions, n distinct hashes, n distinct provenance records. With canonicalization, the naming and structural variation is collapsed (as measured in §4.2), but *intent* divergence—choosing different operations or different parameters—is not addressed by the normalizer.

Specification-level generation. When the generator proposes a typed specification instead of an expression, surface-form divergence is eliminated by construction: the generator never writes the executable expression. The specification names a family and parameter ranges; the system generates the expressions. Divergence can occur only at the specification level: different families, different metrics, different parameter ranges. This is intent divergence, not surface-form divergence.

Divergence comparison. Table 7 compares the divergence sources at each level.

Table 7: Divergence sources under expression-level and specification-level generation. Specification-level generation eliminates surface-form divergence entirely.

Divergence source	Expression	Specification
Operation naming	yes	no
Expression structure (threading)	yes	no
Argument order	yes	no
Family selection	yes	yes
Metric selection	yes	yes
Parameter values	yes	yes

Under expression-level generation, the canonicalization pipeline collapses naming, structural, and argument-order variation. Under specification-level generation, these sources are absent entirely. The remaining divergence (family, metric, parameter selection) is intent divergence: different specifications produce different canonical forms, and the system correctly treats them as distinct.

Combined architecture. BLISP supports both levels. The primary pipeline routes through typed specifications (§3.1): the generator proposes a specification, the system expands it to morphisms. Canonicalization still operates on the expanded expressions (ensuring that the morphism hashes are canonical), but the generator does not produce the expressions. When a generator does produce expressions directly (e.g., for ad-hoc computations not covered by a registered family), the canonicalization pipeline applies. The two mechanisms are complementary: specification-level generation prevents surface-form divergence; canonicalization handles it when it occurs.

5 Related Work

This paper introduces canonical execution semantics for stochastic program generators. Individual components—normalization, content addressing, typed specifications, provenance hashing—exist in other systems. The contribution is their integration around a specific problem: stable execution identity under stochastic generation. We distinguish BLISP from five categories of related work.

5.1 Compiler Normalization and Query Planning

Compilers normalize expressions to improve runtime performance. Constant folding, dead-code elimination, and strength reduction transform programs into faster equivalent forms. Query planners in relational databases apply algebraic rewrites (join reordering, predicate pushdown, subquery flattening) to produce efficient execution plans. Both systems admit multiple valid normal forms: two programs that produce identical outputs may compile to different intermediate representations, and two logically equivalent queries may produce different execution plans, as long as both are correct and fast.

BLISP canonicalizes for a different purpose. The optimization target is not runtime efficiency but execution identity: a unique normal form per computation, so that content-addressed hashing, deduplication, and replay comparison produce stable results under stochastic surface-form variation. Traditional compilers and query planners do not require unique normal forms because their inputs come from human authors who write each expression once. When the generator is stochastic, the same intended computation may arrive in many

surface forms across independent runs. Without a unique canonical form, each variant receives a different hash and a different provenance record.

The rewrite rules in BLISP's normalization algebra (threading expansion, alias resolution, argument reordering) are structurally simple compared to compiler optimizations. This is deliberate: the rewrites exist to collapse surface-form variation, not to improve throughput. IR-level optimization (fusion) is a separate, subsequent pass that does not alter the canonical identity.

5.2 Content-Addressed Systems

Content-addressed build systems hash their inputs to produce deterministic, reproducible outputs. Nix [8] hashes all build inputs—source code, dependencies, compiler flags, and metadata—into a derivation hash. Guix [4] follows the same model. Bazel [3] hashes build actions and their inputs to enable remote caching and reproducible builds. Git [10] content-addresses objects (blobs, trees, commits) by their SHA-1 hash.

BLISP shares the core principle: execution artifacts are identified by content, not by name or path. Three specific differences arise in the context of stochastic generation:

1. **Description/identity separation.** Nix, Guix, and Bazel hash *all* metadata into the derivation identity. Changing a package description, a maintainer field, or a comment in a build file alters the hash and invalidates downstream caches. BLISP separates discovery metadata (DISC) from computational identity (SEM, ALG, IMPL). This allows the registry's natural-language interface to evolve—adding aliases so that stochastic generators discover capabilities under more terms, improving descriptions for better selection—without invalidating prior execution hashes.
2. **Mandatory canonicalization before hashing.** Content-addressed build systems hash their inputs as-is. If two build files are syntactically different but functionally identical, they produce different hashes. BLISP canonicalizes surface forms before hashing, collapsing stochastic variation into a unique canonical form.
3. **Layered provenance decomposition.** Nix produces a single derivation hash per build output. BLISP decomposes the execution hash into 8 semantic layers, enabling per-layer fault localization without re-execution.

These differences are not superiority claims. Nix and Bazel solve build reproducibility; BLISP addresses execution identity under stochastic generation. The problems overlap but are not identical.

5.3 Workflow Systems and Notebooks

Workflow systems (Airflow [1], Dagster [6], Prefect [19]) represent computations as directed acyclic graphs (DAGs) of tasks. They provide scheduling, dependency management, and (with configuration) idempotent re-execution. However, workflow tasks are typically name-addressed (by task ID and DAG ID), not content-addressed. Changing a task's implementation without changing its name does not change its identity in the workflow system. Replay depends on external state management (databases, file systems) rather than content-addressed execution hashing.

Computational notebooks (Jupyter [12], Observable [15]) optimize for interactive, exploratory human workflows. Execution is mutable: cells can be re-run in any order, state accumulates across cells, and the notebook's computational identity is its current cell state rather than a content-addressed hash. Notebooks provide no canonical forms, no replay guarantees, and no provenance decomposition.

The distinction is in the execution model. Workflow systems and notebooks assume human-directed, mutable execution. BLISP assumes stochastically-generated, immutable execution where the primary concern is stable execution identity across independent runs. The two models serve different regimes; BLISP does not replace notebooks for exploratory work, and notebooks do not provide the execution identity guarantees that stochastic generation requires.

5.4 DSLs and Typed Execution Systems

Domain-specific languages provide typed, constrained execution within a specific domain. SQL provides typed relational operations with a query planner. Stan [21] provides a typed probabilistic programming

language with automatic differentiation. dbt provides typed SQL transformations with dependency tracking and testing. Typed APIs (Protocol Buffers, GraphQL) enforce schema constraints on data exchange.

These systems share BLISP's property of typed, constrained execution. The distinction is the assumed generator. SQL, Stan, and dbt assume human authorship: a programmer writes the query or model, and the type system catches errors at development time. BLISP assumes stochastic generation: the generator proposes typed specifications, and the execution system expands them into canonicalized, content-addressed morphisms. The contribution is not the type system or the DSL itself, but the execution semantics that emerge when the generator is stochastic: mandatory canonicalization to collapse surface-form divergence, content-addressed identity for deduplication and replay, and layered provenance for divergence localization.

A reviewer might reasonably ask: "Is this not just a DSL?" The answer is that the normalization algebra, the content-addressed execution hash, the typed morphism expansion, and the description/identity separation are execution mechanisms motivated by stochastic generation. They would be unnecessary overhead in a human-authored DSL, where each expression is written once and stylistic variation is handled by convention.

5.5 Tool Routing and Constrained Generation

Tool-routing systems (ReAct [22], Toolformer [20], Gorilla [18]) teach language models to select and invoke external tools. They address *which* tool to call and *how* to format the call. They do not canonicalize the downstream executable identity, decompose execution provenance, or provide replay guarantees. Two tool invocations with different surface forms but the same intended operation are treated as distinct by the routing system.

Constrained decoding systems (Outlines [17], Guidance [11]) restrict the generator's output to conform to a grammar or schema. This reduces malformed output but does not address surface-form divergence within the valid output space. A constrained generator can still produce multiple valid surface forms for the same canonical computation—it produces fewer invalid forms, but the valid forms still vary.

Retrieval-constrained generation (retrieval-augmented generation, or RAG) conditions the generator on retrieved context. This improves factual grounding but does not canonicalize execution identity. Two retrieval-augmented generations with the same context and prompt can still produce syntactically different expressions.

BLISP operates downstream of all these systems. Tool routing selects *which* capability to invoke. Constrained decoding ensures the output is *well-formed*. Retrieval constrains *what context* the generator uses. BLISP canonicalizes the *executable identity* of the result—collapsing surface-form variation, computing content-addressed hashes, and enabling deterministic replay. These mechanisms are complementary, not competing.

5.6 AI Execution and Agent Orchestration Systems

Recent AI agent frameworks provide orchestration for multi-step LLM-driven computation: routing prompts to tools, managing state across steps, and coordinating multiple agents. These systems address *how* to organize agent execution. They do not address what happens to the executable identity of the computation that agents produce. Table 8 compares execution-identity properties across six agent frameworks and BLISP.

LangGraph [13] (LangChain) provides stateful, graph-based agent orchestration. Agents traverse a state graph with conditional edges. LangGraph addresses execution *flow*—which step runs next, what state is carried—but does not canonicalize the executable identity of the computation that each step produces. Two runs that invoke the same tool with syntactically different arguments produce different execution traces with no mechanism for recognizing canonical equivalence.

AutoGen [2] (Microsoft) provides multi-agent conversation frameworks where agents exchange messages to solve tasks. Execution identity is implicit in the conversation transcript. There is no normalization of tool calls, no content-addressed execution hashing, and no mechanism for determining whether two conversation-driven executions produced canonically equivalent computations.

MCP [14] (Model Context Protocol, Anthropic) standardizes how LLMs connect to external tools and data sources via a JSON-RPC transport. MCP addresses tool *discovery* and *invocation format*—it defines how a model finds and calls a tool. It does not canonicalize the tool call's executable identity, hash the

Table 8: Execution-identity properties across AI agent frameworks. "Canonical form": the system reduces surface-form variation to a unique executable representation. "Content-addressed": execution artifacts are identified by hash, not by name or run ID. "Replay by hash": equivalence is verified by hash comparison without re-execution. "Provenance decomp.": divergence is localizable to a specific semantic layer. "Desc/id sep.": discovery metadata changes do not alter execution identity.

System	*Canonical form*	*Content-addressed*	*Replay by hash*	*Provenance decomp.*	*Desc/id sep.*
LangGraph	–	–	–	–	–
AutoGen	–	–	–	–	–
MCP	–	–	–	–	–
OpenAI Agents SDK	–	–	–	–	–
DSPy	–	–	–	–	–
CrewAI	–	–	–	–	–
BLISP	✓	✓	✓	✓	✓

execution result, or provide replay guarantees. Two MCP tool calls with different argument formatting but the same intended computation are distinct from MCP's perspective.

OpenAI Agents SDK [16] provides agent orchestration with tool use, handoffs between agents, and guardrails. Like LangGraph, it addresses execution flow. Tool invocations are recorded as events but not canonicalized or content-addressed. Replay depends on external state, not on execution identity hashing.

DSPy [9] provides programming abstractions for LLM pipelines: signatures, modules, and optimizers that tune prompts automatically. DSPy addresses *prompt optimization*—finding prompts that produce better outputs—but operates at the generation level, not the execution level. The outputs of DSPy modules are not canonicalized, content-addressed, or decomposed into provenance layers.

CrewAI [5] provides role-based multi-agent orchestration where agents with defined roles collaborate on tasks. Like AutoGen, execution identity is implicit in the agent conversation flow. There is no canonical form for the computation that agents produce, no content-addressed execution hash, and no provenance decomposition.

Distinction. These frameworks solve orchestration: how to route prompts to tools, manage multi-step state, coordinate agents. BLISP solves execution identity: how to ensure that stochastically generated computation has a deterministic, content-addressed, replayable canonical form. The two concerns are complementary. An agent framework could route a prompt to a BLISP execution system, gaining canonical execution identity without changing its orchestration model. The comparison is not "BLISP vs. LangGraph" but "orchestration vs. execution identity"—different layers of the same problem.

6 Discussion and Limitations

Single evaluation domain. All experiments evaluate in one domain: systematic trading research. The architecture (canonicalization pipeline, typed morphism expansion, content-addressed hashing, description/identity separation) is domain-independent, but the evaluation is not. We cannot claim that the measured canonicalization collapse ratios, morphism counts, or alias distributions transfer to other domains. Replication in different domains—bioinformatics pipelines, infrastructure-as-code, data engineering—would strengthen the generality claim.

One generator model. The evaluation uses one stochastic generator (Claude Sonnet). Different generators (GPT-4, open-source models, evolutionary search, program synthesizers) may produce different surface-form distributions, different alias preferences, and different structural patterns. The canonicalization pipeline's effectiveness depends on the overlap between the generator's surface-form vocabulary and the normalization algebra's rewrite rules. A generator that produces forms outside the alias table would not benefit from alias resolution.

No formal semantics. The paper does not provide a formal denotational or operational semantics for BLISP. Canonical equivalence is defined operationally: two expressions are equivalent if they normalize to the same canonical form under the BLISP normalization algebra. This is system-relative, not universal. Two expressions that are equivalent under a richer equational theory (e.g., `(+ x x)` and `(* x 2)`) but not under the rewrite rules will not be identified as equivalent. A formal semantics would enable stronger equivalence claims; its absence is a limitation.

No confluence, no completeness. The normalization algebra is deterministic and idempotent (§3.5), but we do not claim confluence or completeness. Confluence would require that all rewrite sequences converge to the same normal form regardless of application order. Completeness would require that the algebra identifies all extensionally equivalent expressions. Neither is proven, and neither is needed for the paper's claims: the evaluation measures the collapse that the algebra achieves within its rewrite classes, not a theoretical maximum.

No guarantees about correctness or intent. The system guarantees execution identity (same canonical form produces the same execution hash), not scientific correctness or intent fidelity. A canonically well-formed, content-addressable, replayable pipeline may still be the wrong pipeline for the user's goal. Admission control (whether a proposal should be allowed to execute) is handled upstream by grounding gates [7] and is outside the scope of this paper.

No runtime-performance claims. This paper measures execution identity, not throughput. The canonicalization pipeline adds a normalization pass before execution; the content-addressed hashing adds a hashing pass after execution. We do not measure or compare these overheads against systems with different goals (Polars, Pandas, Spark). Runtime performance is orthogonal to the paper's contribution.

Partial coverage by existing systems. Many existing systems already provide components of what BLISP integrates. Compilers normalize. Nix content-addresses. Workflow systems replay. DSLs type-check. The contribution is not any single mechanism but their integration around stochastic execution divergence as a first-class problem. A system that content-addresses but does not canonicalize (Nix) will produce different hashes for syntactically different but functionally identical inputs. A system that canonicalizes but does not content-address (a formatter) will not provide replay guarantees or provenance decomposition. The claim is that stochastic generation creates a design regime where these mechanisms are needed together.

Conservative registry fingerprint. The registry hash h_R is the SHA-256 of all capability identity hashes. Adding an unrelated capability changes h_R and cascades to all execution hashes that include `DIC_HSH`. This is conservative: it invalidates provenance even when the added capability is never used by the pipeline. A finer-grained approach (hashing only the capabilities actually used) would reduce unnecessary invalidation at the cost of additional bookkeeping. This is a known trade-off, not an oversight.

Categorical interpretation. The typed specification can be viewed as an object in a category of constrained parameter spaces, with morphism expansion as a functor to the category of executable configurations, and canonicalization as a natural transformation collapsing surface-form variation. We do not develop this interpretation; it is noted as a possible direction for future formalization that could yield stronger compositional guarantees.

7 Conclusion

This paper treats stochastic divergence as a first-class execution problem. When the generator of computation is stochastic, surface-form variation propagates through hashing, provenance, deduplication, and replay unless the execution system intervenes. We argued that execution systems for stochastic generators require a canonical execution boundary: an architectural invariant beyond which stochasticity does not propagate.

We reported on BLISP, a deterministic execution system that enforces this boundary through four mechanisms:

1. **Typed specification as constrained proposal.** Stochastic generators propose declarative parameter tuples from a constrained semantic space; the system expands these into canonicalized, content-addressed morphisms via Cartesian product. The generator never writes the executable expression, eliminating surface-form divergence at the expression level by construction.
2. **Canonical execution identity.** A traced normalization algebra collapses remaining surface-form divergence to unique canonical forms. In the static registry, 278 accepted surface forms collapse to 235 canonical operations (ratio 1.18). After canonicalization, all downstream execution artifacts are deterministic functions of the canonical form, registry state, and input data.
3. **Layered provenance hashing.** An 8-layer execution hash decomposes end-to-end provenance into distinct semantic layers. When two execution hashes differ, sub-hash comparison localizes the divergence without re-execution. Replay equivalence is verified by hash comparison alone.
4. **Description/identity separation.** Capability identity is content-addressed over three computational layers, with discovery metadata excluded. The registry's natural-language interface evolves independently of execution identity, resolving the tension between discoverability and reproducibility.

The evaluation measured canonicalization collapse under 1,200 stochastic LLM generations, replay equivalence across 50 independent runs, provenance stability under registry evolution, morphism deduplication, and divergence properties at the specification level versus direct expression generation. The results are consistent with the central thesis: AI-generated computation should not execute stochastic outputs directly; stochastic proposals should first compile into deterministic semantic objects before execution begins.

These mechanisms are not individually novel. Compilers normalize. Content-addressed systems hash. Typed languages constrain. The contribution is their integration around a canonical execution boundary that partitions the pipeline into a stochastic upstream and a deterministic downstream. This is a problem that arises when the generator of computation is stochastic rather than human, and that existing programming systems—including current AI agent frameworks—address partially at best.

The broader implication is architectural: AI systems that generate computation should compile stochastic proposals into deterministic semantic objects—typed specifications with canonical forms, content-addressed hashes, and decomposed provenance—before execution begins. The canonical execution boundary is not a BLISP-specific mechanism; it is a design principle for any execution system whose generator is non-deterministic. The specific mechanisms will vary across domains; the invariant—that stochasticity does not propagate beyond canonicalization—should not. The architecture is evaluated in one domain with one generator; broader evaluation across domains and generators is future work.

References

[1] Apache Software Foundation. Apache Airflow. `https://airflow.apache.org`, 2015.

[2] Q. Wu et al. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv:2308.08155*, 2023.

[3] Google. Bazel: a fast, scalable, multi-language build system. `https://bazel.build`, 2015.

[4] L. Courtès. Functional package management with Guix. In *European Lisp Symposium*, 2013.

[5] CrewAI, Inc. CrewAI: Framework for orchestrating role-playing, autonomous AI agents. `https://github.com/crewAIInc/crewAI`, 2024.

[6] Elementl. Dagster: the data orchestration platform. https://dagster.io, 2019.

[7] T. Dionysopoulos. The grounding gate: Admissibility and replay guarantees for AI-driven research. *Zenodo*, 2026. doi:10.5281/zenodo.20456984.

[8] E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In *LISA*, 2004.

[9] O. Khattab et al. DSPy: Compiling declarative language model calls into self-improving pipelines. In *ICLR*, 2024.

[10] L. Torvalds. Git: a distributed version control system. https://git-scm.com, 2005.

[11] Microsoft. Guidance: a guidance language for controlling large language models. https://github.com/guidance-ai/guidance, 2023.

[12] T. Kluyver et al. Jupyter notebooks – a publishing format for reproducible computational workflows. In *ELPUB*, 2016.

[13] LangChain, Inc. LangGraph: Build stateful, multi-actor applications. https://github.com/langchain-ai/langgraph, 2024.

[14] Anthropic. Model Context Protocol. https://modelcontextprotocol.io, 2024.

[15] Observable, Inc. Observable: the JavaScript notebook. https://observablehq.com, 2018.

[16] OpenAI. OpenAI Agents SDK. https://github.com/openai/openai-agents-python, 2025.

[17] B. T. Willard and R. Louf. Efficient guided generation for large language models. *arXiv:2307.09702*, 2023.

[18] S. G. Patil et al. Gorilla: Large language model connected with massive APIs. *arXiv:2305.15334*, 2023.

[19] Prefect Technologies. Prefect: the modern data stack workflow system. https://www.prefect.io, 2019.

[20] T. Schick et al. Toolformer: Language models can teach themselves to use tools. In *NeurIPS*, 2023.

[21] B. Carpenter et al. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(1), 2017.

[22] S. Yao et al. ReAct: Synergizing reasoning and acting in language models. In *ICLR*, 2023.